From the 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM), Beijing, China, September 2004, IEEE Computer Society, pp. 261–270.

Generating Efficient Test Sets with a Model Checker[*]

Grégoire Hamon
Computing Science Department
Chalmers University of Technology
Göteborg, Sweden
hamon@cs.chalmers.se

Leonardo de Moura, and John Rushby
Computer Science Laboratory,
SRI International
Menlo Park CA USA
demoura|rushby@csl.sri.com

Abstract

It is well-known that counterexamples produced by model checkers can provide a basis for automated generation of test cases. However, when this approach is used to meet a coverage criterion, it generally results in very inefficient test sets having many tests and much redundancy. We describe an improved approach that uses model checkers to generate efficient test sets. Furthermore, the generation is itself efficient, and is able to reach deep regions of the statespace. We have prototyped the approach using the model checkers of our SAL system and have applied it to model-based designs developed in Stateflow. In one example, our method achieves complete state and transition coverage in a Stateflow model for the shift scheduler of a 4-speed automatic transmission with a single test case.

1. Introduction

Automated generation of test cases is an attractive application for mechanized formal methods: the importance of good test cases is universally recognized, and so is the high cost of generating them by hand. And automated test generation not only provides an easily perceived benefit, but it is becoming practical with current technology and fits in with established practices and workflows.

We focus on reactive systems (i.e., systems that constantly interact with their environment), where a test case is a sequence of inputs from its environment that will cause the system under test to exhibit some behavior of interest. To perform the tests, the system is combined with a test harness that simulates its environment; the test harness initiates and engages in an interaction with the system that guides it through the intended test case and observes its response. For simplicity of exposition, we will assume that the test harness has total control of the environment and that the system under test is deterministic.

An effective approach to automated test generation is based on the ability of model checkers to generate counterexamples to invalid assertions: roughly speaking, to generate a test case that will exercise a behavior characterized by a predicate p, we model check for the property "always not p" and the counterexample to this property provides the required test case (if there is no counterexample, then the property is true and the proposed test case is infeasible). This approach seems to have been first applied on an industrial scale to hardware [10] and on a more experimental scale to software [4], although related technologies based on state machine exploration have long been known in protocol testing [20].

Generally, individual test cases are generated as part of a *test set* designed to achieve some desired *coverage* and there are two measures of cost and efficiency that are of interest: what is the cost to *generate* a test set that achieves the coverage target (this cost is primarily measured in CPU time and memory, and may be considered infeasible if it goes beyond a few hours or requires more than a few gigabytes), and what is the cost to *execute* the test set that is produced? For execution, an efficient test set is one that minimizes the number of tests (because in executing the tests, starting a new case can involve fairly costly initialization activities such as resetting attached hardware), and their total length (because in executing tests, each step exacts some cost). Many methods based on model checking generate very inefficient test sets: for example, they generate a separate test for each case to be covered, and the individual tests can be long also. This paper is concerned with methods for generating test sets that are efficient with respect to both generation and execution. Section 2 introduces our methods, which work by iteratively extending already discovered tests so that they discharge additional goals.

[*] This research was supported in part by NASA Langley Research Center contract NAS1-00079, by National Science Foundation grant CCR-00-86096, by the Swedish Foundation for Strategic Research, and by SRI International.

The feasibility and cost of generating test sets are obviously dependent on the underlying model checking technology. The worst-case complexity of model checking is linear in the size of the reachable state space (the "state explosion problem" recognizes that this size is often exponential in some parameter of the system), but this complexity concerns valid assertions, whereas for test generation we use deliberately invalid assertions and the time to find a counterexample, while obviously influenced by the size of the statespace, is also highly sensitive to other attributes of the system under examination, to the test cases being sought, and to the particular technology and search strategy employed by the model checker. Any given model checking method is very likely to run out of time or memory while attempting to generate some of the test cases required for coverage; Section 3 of the paper discusses the pragmatics of model checking for the purpose of test generation.

We believe that the methods we present will be effective for many kinds of system specifications, and for many notions of coverage, but our practical experience is with model-based development of embedded systems. Here, executable models are constructed for the system and its environment and these are used to develop and validate the system design. The model for the system then serves as the specification for its implementation (which is often generated automatically). The model is usually represented in a graphical form, using statecharts, flowcharts, message sequence charts, use diagrams, and so on. Most of our experience is with Stateflow [19], which is the combined statechart and flowchart notation of Matlab/Simulink, the most widely used system for model-based design. Section 4 of the paper describes the results of some modest experiments we have performed using our method.

1.1. Background and terminology

Coverage is often specified with respect to the *structure* of a design representation: in this context, *state coverage* means that the test set must visit every control location in the representation, while *transition coverage* means that the test set must traverse every transition between control locations. For certain safety-critical applications, a rather exacting type of coverage called modified condition/decision coverage (MC/DC) is mandated. It is usually required that test coverage is measured and achieved on the *implementation*, but that the test cases must be generated by consideration of its functional *requirements* (see [12]). An approach that is gaining popularity in model-based design is to generate test sets automatically by targeting structural coverage in the representation of the model: the intuition is that if we generate tests to achieve (say) transition coverage in the model, then that test set is very likely to come close to achieving transition coverage in the implementation. This approach interprets the model as representing the functional requirements (it also serves as the oracle for evaluating test outcomes); a variation (used for example by Motorola in its VeriState tools[1]) augments the model with requirements and test "observers" and targets structural coverage on these.

In practical terms, automated test generation proceeds by translating the system model into the language of a model checker, then constructing assertions whose counterexamples, when "concretized" to the form required by the implementation to be tested, will provide the desired coverage. The assertions are typically temporal logic formulas over "trap properties" [9] that characterize when execution of the system reaches a certain control point, takes a certain transition, or exhibits some other behavior of interest. Trap properties can be expressed in terms of state variables that are inherent to the representation, or the translation to the language of the model checker can introduce additional state variables to simplify their construction. Most of the following presentation is independent of the particular notion of coverage that is selected and of the method for constructing trap properties and their associated temporal logic assertions. We will, however, speak of the individual cases in a coverage requirement as test *goals* (so the requirement to exercise a particular transition is one of the test goals within transition coverage).

2. Efficient tests by iterated extension

The basic problem in the standard approach to test generation by model checking is that a separate test case is generated for each test goal, leading to test sets having much redundancy. We can illustrate this problem in the example shown in Figure 1, which presents the Stateflow specification for a stopwatch with lap time measurement.

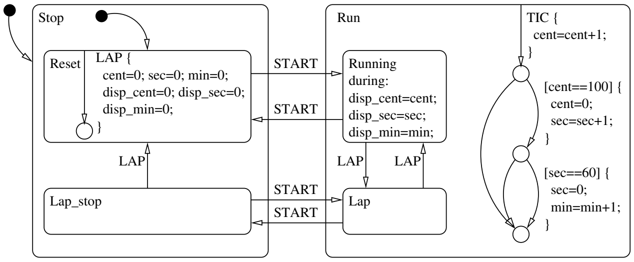

Figure 1: A simple stopwatch in Stateflow

[1] See www.motorola.com/eda/products/veristate/veristate.html.

The stopwatch contains a counter represented by three variables (min, sec, cent) and a display, also represented as three variables (disp_min, disp_sec, disp_cent).

The stopwatch is controlled by two command buttons, START and LAP. The START button switches the time counter on and off; the LAP button fixes the display to show the lap time when the counter is running and resets the counter when the counter is stopped. This behavior is modeled as a statechart with four exclusive states:

- Reset: the counter is stopped. Receiving LAP resets the counter and the display, receiving START changes the control to the Running mode.
- Lap_Stop: the counter is stopped. Receiving LAP changes to the Reset mode and receiving START changes to the Lap mode.
- Running: the counter is running, and the display updated. Receiving START changes to the Stop mode, pressing LAP changes to the Lap mode.
- Lap: the counter is running, but the display is not updated, thus showing the last value it received. Receiving START changes to Lap_Stop mode, receiving LAP changes to theRunning mode.

These four states are grouped by pairs inside two main states: Run and Stop, active when the counter is counting or stopped, respectively. The counter itself is specified within the Run state as a flowchart, incrementing its cent value every time a clock TIC is received (i.e., every 1/100s); the sec value is incremented (and cent reset to 0) whenever cent equals 100, and the min value is similarly incremented whenever sec equals 60.

Notice that it requires a test case of length 6,000 to exercise the lower right transition in the flowchart: this is where the min variable first takes a nonzero value, following 60 secs, each of 100 cents. Embedded systems often contain counters that must be exhausted before parts of the statespace become reachable so this is a (perhaps rather extreme) example of the kind of "deep" test goal that is often hard to discharge using model checking.

Focusing now on the statechart to the left of the figure, if we generate a test case that begins in the initial state and exercises the transition from Lap_stop to Reset (e.g., the sequence of events START, LAP, START, LAP), then this test also exercises the transitions from Reset to Running, Running to Lap, and Lap to Lap_stop. However, the usual approach to generating a test set to achieve transition coverage will independently generate test cases to exercise each of these transitions, resulting in four tests and much redundancy. Black and Ranville [3] describe a method for "winnowing" test sets after generation to reduce their redundancy, while Hong et al. [16] present an algorithm that reduces redundancy during generation. Their algorithm will record during generation of a test case to exercise the Lap_stop to Reset transition that it has also exercised the Running to Lap transition and will remove the latter transition from its set of remaining coverage goals. However, the effectiveness of this strategy depends on the order in which the model checker tackles the coverage goals: if it generates the test for Running to Lap before the one for Lap_stop to Reset, then this online winnowing will be ineffective.

A natural way to overcome this inefficiency in test sets is to attempt to *extend* existing test cases to reach uncovered goals, rather than start each one afresh. This should not only eliminate much redundancy from the test set, but it should also reduce the total number of test cases required to achieve coverage. Although conceptually straightforward, it is not easy in practice to cause a model checker to find a counterexample that extends an existing one when the only way to interact with the model checker is through its normal interfaces (where all one can do is supply it with a system specification, an initial state, and a property). Fortunately, several modern model checkers provide more open environments than was previously the case; in particular, they provide scriptable interfaces that permit rapid construction of customized analysis tools.

We performed our experiments in the SAL 2 model checking environment [6], which not only provides state-of-the-art symbolic, bounded, infinite-bounded, and witness model checkers, but also an API that gives access to the basic machinery of these tools and that is scriptable in the Scheme language [17] (in fact, the model checkers are themselves just Scheme scripts).[2] Among the API functions of SAL 2, or easily scripted extensions to these, are ones to perform a (symbolic or bounded) model check on a given system and property, and to continue a model check given a previously reached state and a path to get there.

Given these API functions, it is easy to construct a script that extends each test case to discharge as many additional coverage goals as possible, and that starts a new test case only when necessary. A pseudocode rendition of this script is shown in Figure 2. On completion, the variable *failures* contains the set of coverage goals for which the algorithm was unable to generate test cases.

It might seem specious (in the most deeply nested part of Figure 2) to remove from *remaining* and *failures* any goals discharged by extending a test case—because this set contains only those that were not discharged by previous attempts to extend the current case. However, if the model checker is using limited resources (e.g., bounded model checking to depth k), a certain goal may be discharged by

[2] We also use the explicit-state model checker of SAL 1, which is distinct from SAL 2, and not completely compatible with it; a future release of SAL will unify these two systems.

```
goals := the set of coverage goals
failures := empty set
while goals is nonempty do
Select and remove goal from goals
Call model checker to generate
 a new test case to discharge goal
if successful then
  Select and remove from goals any that
    are discharged by the test case
  remaining := empty set
  while goals is nonempty do
    Remove goal from goals
    Call model checker to extend
      test case to discharge goal
    if successful then
      remove from goals, failures, and
       remaining any goals
       discharged by extended test case
    else add goal to remaining
    endif
  endwhile
  goals := remaining
  Output test case
else add goal to failures endif
endwhile
```

Figure 2: Constructing test cases by iterated extension

an extension that can be found by model checking from a given test case, but not from its prefixes.

Although quite effective, the method of Figure 2 fails to exploit some of the power of model checking: at each step, it selects a particular coverage goal and tries to discharge it by generating a new test case or extending the current one. This means that the coverage goals are explored in some specific order that is independent of their "depth" or "difficulty."

It actually improves the speed of model checking if we consider multiple goals in parallel: instead of picking a goal and asking the model checker to discharge it, we can give it the entire set of undischarged goals and ask it to discharge any of them. That is, instead of separately model checking the assertions "always not p," "always not q" etc., we model check "always not (p or q or…)." This will have the advantage that the model checker will (probably) first discharge shallow or easy goals and approach the deeper or more difficult goals incrementally; as noted above, it may be possible to discharge a difficult goal by extending an already discovered test case when it could not be discharged (within some resource bound) from an initial state, or from a shorter test case generated earlier in the process.

A further refinement is to note that as test generation proceeds, those parts of the system specification that have already been covered may become irrelevant to the coverage goals remaining. Modern model checkers, including SAL, generally perform some form of automated *model reduction* that is similar to (backward) program slicing [21]. Typically, they use the *cone of influence reduction* [18]: the idea is to eliminate those state variables, and those parts of the model, that do not influence the values of the state variables appearing in the assertion to be model checked.

If we use this capability to slice away the parts of the system specification that become irrelevant at each step then the specification will get smaller as the outstanding coverage goals become fewer. Notice there is a virtuous circle here: slicing becomes increasingly effective as the outstanding goals become fewer; those outstanding goals are presumably hard to discharge (since the easy ones will be picked off earlier), but slicing is reducing the system and making it easier to discharge them. Recall that in Figure 1 it requires a test case of length 6,000 to exercise the lower right transition in the flowchart. There is almost no chance that a model checker could quickly find the corresponding counterexample while its search is cluttered with the vast number of display and control states that are independent of the state variables representing the clock. Once the coverage goals in the statechart part of the model have been discharged, however, all those state variables can be sliced away, isolating the flowchart and rendering generation of the required counterexample feasible (we present data for this example later). Pseudocode for this refinement to the method is shown in Figure 3.

Still further improvements can be made in this approach to generating test sets. The method of Figure 3 always seeks to extend the current test case, and if that fails it starts a new case. But the test cases that have already been found provide the ability to reach many states, and we may do better to seek an extension from some intermediate point of some previous test case, rather then start a completely new case when the current case cannot be extended. This is particularly so when we have already found one deep test case that gives entry to a new part of the statespace: there may be many coverage goals that can be discharged cheaply by constructing several extensions to that case, whereas the method of Figure 3 would go back to the initial state once a single extension to the test case had been completed.

Figure 4 presents pseudocode for a search method that attempts (in the nested **while** loop) to extend the current test case as much as possible, but when that fails it tries (in the outer **while** loop) to extend a test from some state that it has reached previously (these are recorded in the variable *knownstates*). Notice that it is not necessary to call the model checker iteratively to search from each of the *knownstates*: a model checker (at least a symbolic or bounded

```
goals := the set of coverage goals
failures := empty set
while goals is nonempty do
Call model checker to generate
 a new test case to discharge some goal
if successful then
  Remove from goals any that
    are discharged by the test case
  slice system relative to goals
  while goals is nonempty do
    Call model checker to extend
      test case to discharge some goal
    if successful then
      remove from goals any
       discharged by extended test case
      slice system relative to goals
    endif
  endwhile
  Output test case
else
  failures := goals;
  goals := empty set
endif
endwhile
```

Figure 3: Searching for test cases in parallel, and slicing the model as goals are discharged

model checker) can search from all these states in parallel. This parallel search capability increases the efficiency of test generation but might seem to conflict with our desire for efficient test sets: the model checker might find a long extension from a known shallow state rather than a short extension from a deeper one. To see how this is controlled, we need to examine the attributes of different model checking technologies, and this is the topic of the next section.

3. Finding the extensions: model checking pragmatics

All model checkers (of the kind we are interested in) take as their inputs the transition relation defining a state machine and its environment, the initial states, and an assertion. The assertion is usually expressed as a temporal logic formula but we are interested only in formulas of the kind "always not p," so the details of the temporal logic are not important. And although the model checker may actually work by encoding the assertion as a Büchi automaton, it does little harm in this particular case to think of the model checker as working by searching for a state that satisfies p and is reachable from the initial states.

```
goals := the set of coverage goals
knownstates := initial states
failures := empty set
while goals is nonempty do
Call model checker to extend a test
 case from some state in knownstates
 to discharge some goal
if successful then
  Remove from goals any that
    are discharged by the test case
  add to knownstates those states
   traversed by the current test case
  slice system relative to goals
  while goals is nonempty do
    Call model checker to extend
      test case to discharge some goal
    if successful then
      remove from goals any
       discharged by extended test case
      add to knownstates those states
       traversed by current test case
      slice system relative to goals
    endif
  endwhile
  Output test case
else
  failures := goals;
  goals := empty set
endif
endwhile
```

Figure 4: Restarting from previously discovered states rather than initial states

The earliest model checkers used an approach now called *explicit state* exploration, and this approach is still very competitive for certain problems. As the name suggests, this kind of model checker uses an explicit representation for states and enumerates the set of reachable states by forward exploration until either it finds a violation of the assertion (in which case a trace back to the start state provides a counterexample), or it reaches a fixed point (i.e., has enumerated all the reachable states without discovering a violation, in which case the assertion is valid).

There are several strategies for exploring the reachable states: *depth first* search uses the least memory and often finds counterexamples quickly, but the counterexamples may not be minimal; *breadth first* search, on the other hand, requires more memory and often takes longer, but will find the shortest counterexamples. Gargantini and Heitmeyer [9] report that counterexamples produced by an explicit-state model checker using depth-first search were often too long to be useful as test cases. Using a translation into SAL for the example of Figure 1, SAL's explicit-state model checker operating in depth-first mode finds a test case for the transition at the bottom right in 25 seconds (on a 2GHz Pentium with 1 GB of memory) after exploring 71,999 states, but the test case is 24,001 steps long. This is 4 times the minimal length because several START and LAP events are interspersed between each TIC. In breadth-first mode, on the other hand, the model checker does not terminate in reasonable time.[3] However, if we slice the model (thereby eliminating START and LAP events), both breadth- and depth-first search generate the minimal test case of length 6,001 in little more than a second.

In summary, explicit-state model checking needs to use breadth-first search to be useful for test case generation, and the search becomes infeasible when the number of states to be explored exceeds a few million; within this constraint, it is capable of finding deep test cases.

For embedded systems, a common case where the reachable states rapidly exceed those that can be enumerated by an explicit-state model checker is one where the system takes several numerical inputs from its environment. In one example from Heimdahl et al. [13], an "altitude switch" takes numerical readings from three altimeters, one of which may be faulty, and produces a safe consensus value. If the altimeters produce readings in the range 0...40,000 feet, then an explicit-state model checker could blindly enumerate through a significant fraction of the $40,000^3$ (i.e., 64 trillion) combinations of input values before stumbling on those that trigger cases of interest. In practice, this simple type of problem is beyond the reach of explicit-state model checkers.

Symbolic model checkers, historically the second kind to be developed, deal with this type of problem in fractions of a second. A symbolic model checker represents sets of states, and functions and relations on these, as reduced ordered binary decision diagrams (BDDs). This is a compact and canonical symbolic representation on which the image computations required for model checking can be performed very efficiently. The performance of symbolic model checkers is sensitive to the size and complexity of the transition relation, and to the size of the total statespace (i.e., the number of bits or BDD variables needed to represent a state), but it is less sensitive to the number of reachable states: the symbolic representation provides a very compact encoding for large sets of states.

Symbolic model checkers can use a variety of search strategies and these can have dramatic impact when verifying valid assertions: for example, backward search verifies inductive properties in a single step. In test generation, however, where we have deliberately invalid properties, a symbolic model checker, whether going forward or backward, must perform at least as many image computations as there are steps in the shortest counterexample. The symbolic model checker of SAL 2 can find the counterexample of length 6,000 that exercises the lower right transition of the flowchart in Figure 1 in 125 seconds (it takes another 50 seconds to actually build the counterexample) and visits 107,958,013 states. If we slice the model (eliminating START and LAP events), then the number of visited states declines to 6,001 and the time decreases to 85 seconds (plus 50 to build the counterexample).

Thus a symbolic model checker can be very effective for test case generation even when there are large numbers of reachable states, and also for fairly deep cases. Its performance declines when the number of BDD variables grows above a few hundred, and when the transition relation is large: both of these increase the time taken to perform image computations, and thus reduce the depth of the test cases that can be found in reasonable time. There is an additional cost to systems that require many BDD variables, and this is the time taken to find a good variable ordering (the performance of BDD operations is very dependent on arranging the variables in a suitable order). Heimdahl et al. [14] report that the time taken to order the BDD variables became the dominant factor in their larger examples, and caused them to conclude that symbolic model checking is unattractive for test generation. Modern model checkers such as SAL 2 alleviate this concern a little: they allow the variable ordering found in one analysis to be saved and reused for others—this amortizes the cost of variable ordering over all tests generated (provided the one ordering is effective for them all). The SAL 2 symbolic model checker also has a mode where it computes the reachable states just once, and then analyzes many safety properties against it.

Bounded model checkers, the third kind to be developed, are specialized to generation of counterexamples (though they can be used to perform verification by k-induction [8]). A bounded model checker is given a depth bound k and searches for a counterexample up to that depth (i.e., length) by casting it as a constraint satisfaction problem: for finite state systems, this can be represented as a propositional satisfiability problem and given to a SAT solver. Modern SAT solvers can handle problems with many thousands of variables and constraints. Each increment of 1 in the depth of bounded model checking increases the number of variables in the SAT problem by the number of bits needed to represent the statespace and by the number of constraints needed to represent the transition relation: empirically, the complexity of bounded model checking is strongly dependent on the depth, and the practical limit on k is around 30–

[3] If we reduce the number of cents in a sec from 100 to 4 (resp. 5), then the breadth-first search terminates in 89 (resp. 165) seconds after exploring 171,133 (resp. 267,913) states; the time required is exponential in this parameter.

50. At modest depths, however, bounded model checking is able to handle very large statespaces and does not incur the startup overhead of BDD ordering encountered in symbolic model checking large systems (though it does have to compute the k-fold composition of the transition relation). It should be noted that a bounded model checker does not necessarily generate the shortest counterexamples: it simply finds some counterexample no longer than k. Obviously, it will find the shortest counterexample if it is invoked iteratively for $k = 1, 2, \ldots$ until a counterexample is found but most bounded model checkers do not operate incrementally, so this kind of iteration is expensive.

Bounded model checking can be extended to infinite state systems by solving constraint satisfaction problems in the combination of propositional calculus and the theories of the infinite data types concerned (e.g., real and integer linear arithmetic). SAL 2 has such an "infinite bounded" model checker; this is based on the ICS decision procedure [5], which has the best performance of its kind for many problems [7]. However, this model checker does not yet produce concrete counterexamples (merely symbolic ones), so we have not used it in our test generation exercises.

Given these performance characteristics of various model checking technologies, which is the best for test case generation? Recall that Gargantini and Heitmeyer [9] report dissatisfaction with unnecessarily long test sequences produce by an explicit-state model checker operating depth first, and satisfaction with a symbolic model checker. On the other hand, Heimdahl et al. [14] report dissatisfaction with a symbolic model checker because of the lengthy BDD ordering process required for large models, and satisfaction with a bounded model checker, provided it was restricted to very modest bounds (depth 5 or so). The examples considered by Heimdahl et al. were such that coverage could be achieved with very short tests, but this will not generally be the case, particularly when counters are present.

Our experiments with the approaches to iterated extension described in the previous section confirm the effectiveness of bounded model checking for test generation. Furthermore, our approach minimizes its main weakness: whereas bounded model checking to depth 5 will not discharge a coverage goal that requires a test case of length 20, and bounded model checking to depth 20 may be infeasible, iterated bounded model checking to depth 5 may find a path to one goal, then an extension to another, and another, and eventually to the goal at depth 20—because four or five checks to depth 5 are much easier than one to depth 20.

However, bounded model checking to modest depths, even when iterated, may be unable to exhaust a loop counter, or to find entry to other deep parts of a statespace. We have found that an effective combination is to use symbolic model checking (with some resource bound) as the model checker at the top of the outer **while** loop in Figure 3. This call is cheap when many easy goals remain (the cost of BDD ordering is amortized over all calls), and can be useful in finding a long path to a new part of the state space when all the easy goals have been discharged. As noted in the previous section, slicing can be especially effective in this situation.

Although we have not yet performed the experiments, we believe that using symbolic model checking in the outer **while** loop in the method of Figure 4 will be an even more effective heuristic. As in Figure 3, using a symbolic model checker in this situation preserves the possibility of finding long extensions, should these be necessary. Equally important, the representation of *knownstates* as a BDD for symbolic model checking is likely to be compact, whereas its representation as SAT constraints for a bounded model checker could be very large. We also conjecture that explicit-state model checking may be useful for finding long paths in heavily sliced models, but it is perhaps better to see this as an instance of a more general approach, developed in the following paragraphs, rather than as an independently useful combination.

All the enhancements to test generation that we have presented so far have used model checking as their sole means for constructing test cases, but there is a natural generalization that leads directly to an attractive integration between model checking and other methods.

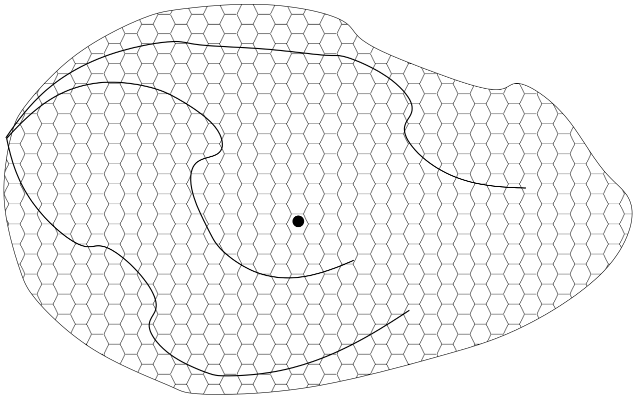

Figure 5: Generalization: *knownstates* seeded by random testing or other methods

In particular, the method of Figure 4 uses the states in the set *knownstates* as starting points for extending known paths into test cases for new goals. As new test cases generate paths to previously unvisited states, the method adds these to *knownstates*, but it starts with this set empty. Suppose instead that we initialize this set with some sampling of states, and the paths to reach them, as portrayed in Figure 5 (the shaded figure suggests the reachable statespace and the three interior lines represent known paths through a sampling of states). Random testing is one way to create the initial population of states and paths, and (concretized) states and paths found by model checking abstractions of the original system could be another (explicit-state model checking in heavily sliced models would be an instance of this). Now, given a goal represented by the solid dot in Figure 5, the method of Figure 4 will start symbolic model checking from all the *knownstates* in parallel and is likely to find a short extension from one of them to the desired goal. If *knownstates* is considered too large to serve as the starting point for model checking, then some collection of the most likely candidates can be used instead (e.g., those closest to the goal by Hamming distance on their binary representations). Of course, if there is more than a single outstanding goal, the symbolic model checker will search in parallel from all *knownstates* to all outstanding goals; once an extension has been found, the bounded model checker will seek to further extend that path; when that path is exhausted the search will revert to the symbolic model checker of the outer loop.

This combination of methods is actually an elaboration of those used in two commercial tools. Ketchum (aka. FormalVera and Magellan) from Synopsys [15] uses bounded model checking to extend paths found by random testing in hardware designs, while Reactis from Reactive Systems Inc.[4] uses constraint solving (similar to the technology underlying infinite bounded model checking) to extend paths found by random testing in Simulink and Stateflow models. Neither of these tools (to our knowledge) uses model checking to search toward the goal from the whole set of *knownstates* (or a large subset thereof); instead they pick a state that is "close" (e.g., by Hamming distance) to the goal. Neither do they use the model checker to search toward all outstanding goals simultaneously.

4. Experimental results

We have implemented the test generation method of Figure 3 as a script that runs on the API of SAL 2.[5] The SAL API is provided by a program in the Scheme language [17] that uses external functions (mostly written in C) to provide efficient implementations of the core model checking algorithms. Our test generation script is thus a small collection of function definitions in Scheme; arguments to the top-level function determine whether or not slicing is to be performed, whether initial searches from the start states should use symbolic or bounded model checking (and, in the latter case, to what depth), and the depth of bounded model checking to be used in the iterated extensions. Despite all these options to support experimentation, the script is less than 100 lines long. The script and the examples described below can be downloaded from `http://www.csl.sri.com/˜rushby/abstracts/sefm04`, where a longer version of this paper also provides extended discussion of the examples.

4.1. Stopwatch

Our first example is the Stopwatch of Figure 1. We have already reported statistics from initial experiments on this example; here, we present data from our completed SAL script. Our script operated on a SAL translation of Figure 1 that was constructed by hand. We targeted state and transition coverage and augmented the SAL specification with a new Boolean trap variable for each coverage goal; the trap variable latches `TRUE` whenever its corresponding coverage goal is satisfied. The trap variables obviously increase the size of the transition relation and system state that must be represented in the model checker, but they play such a passive role that they do not add an appreciable burden to the model checking process.

The shortest test needed for full state and transition coverage in the statechart part of Figure 1 is 11 steps. When tests are generated separately for each coverage goal, our script produces 11 separate tests with a total length of 26. When the optimization of Hong et al. is enabled (i.e., we check to see whether a newly generated test happens to discharge goals other than the one targeted), the test set is reduced to 8 tests with a total length of 20 steps. And when the iterated extension method of Figure 3 is enabled, our script achieves coverage with a single test of between 11 and 14 steps (depending on the model checking parameters used). When the flowchart part of Figure 1 is added to the coverage goals, our script generates three tests: one of length 12 that covers the statechart part of the figure, one of length 101 that exercises the rollover of the `cent` variable from 99 to 0, and one of length 6001 that exercises the rollover of the `sec` variable from 59 to 0 (the second test is subsumed by the third but our method does not detect this). Slicing ensures that the second and third tests are generated in a reduced model in which the variables corresponding to the statechart part of the program have been removed, and the generation is therefore quite efficient (e.g., it takes 84 seconds to generate the third, and longest test). Slicing can be disabled for experimental purposes; doing so in this example increases the generation time fivefold.

[4] See `www.reactive-systems.com`.

[5] We are in the process of implementing the method of Figure 4, which requires some extensions to the API.

4.2. Shift Scheduler

Our next example is a shift scheduler for a four-speed automatic transmission that was made available by Ford as part of the DARPA MoBIES project.[6] The Stateflow representation of this example has 23 states and 25 transitions. We converted the Stateflow component of the Matlab `.mdl` file for this example into SAL using a prototype translator based on the Stateflow semantics presented in [11] that automatically adds trap variables to latch state and transition coverage goals. (A couple of internal names were changed by hand as our translator does not yet handle all the naming conventions of Matlab.) Several of the inputs to this example are real numbers; we changed them to 8-bit integers for model checking purposes. The Stateflow component of the Matlab model is not really a self-contained unit: it has six inputs `shift_speed_ij` (these determine when a shift from gear i to j should be scheduled) that are not independent and are driven from a single `torque` input in the larger Simulink block that surrounds the Stateflow model. We do not have a translator from Simulink to SAL so we constructed a suitable SAL rendition for this part of the model by hand (it is just a dozen lines of elementary SAL that tie a `torque` input to the `shift_speed_ij` variables). The composition of these two SAL modules has 288 state bits in the representation used by the model checker. (300 state bits is generally regarded as the point where model checking can become difficult.) Using iterated extension, our script generated a single test in a couple of minutes that achieves full state and transition coverage in 73 steps.

4.3. Flight Guidance System

Our final example is a model of an aircraft flight guidance system developed by Rockwell Collins under contract to NASA for the purpose of aiding experiments such as this [14]. The models were originally developed in RSML; we used SAL versions kindly provided by Jimin Gao of the University of Minnesota who is developing an RSML to SAL translator. The largest of the examples is `ToyFGS05_Left`, which has 576 state variables and requires 1,152 BDD variables for symbolic model checking. The SAL version of this specification is not instrumented with trap variables for coverage, but the model does contain 369 Boolean variables. We conjecture that for the purposes of measuring performance and scaling effects, generating tests to drive these variables to `TRUE` will be an effective experiment. Many of the Boolean variables have names ending in `_Undefined` or `_Random` and seem to be present for error detection and not intended to become activated. We eliminated these and targeted the remaining 185 variables. In less than five minutes, our script succeeded in building a single test case of length 44 that takes all but two of the Boolean state variables to `TRUE` (we separately verified that those two variables are invariantly `FALSE`). Slicing is fairly effective in this example as the model checking problem is reduced from 576 Boolean variables at the start to 421 at the end, and the overall time taken is halved when slicing is used.

5. Summary and future plans

We have described a method for generating efficient test sets for model-based embedded systems by using a model checker to extend tests discovered earlier in the process. Extending tests not only eliminates the redundancy of many tests with similar prefixes, but it allows the model checker incrementally to explore deeper into the statespace than might otherwise be possible within given resource bounds, leading to more complete coverage. Our method requires "going under the hood" of the model checker to exploit the capabilities of its API, but several modern model checkers provide a suitably scriptable API. Our methods exploit the full power of model checking to search at each step for an extension from any known state to any uncovered goal, and use slicing so that the complexity of the system being model checked is reduced as the outstanding coverage goals become harder to achieve. We described how the method can be combined with others, such as random testing, that create a preliminary "map" of known paths into the statespace.

We discussed the pragmatics of different model checking techniques for this application and described preliminary experiments with the model checkers of our SAL system. Our preliminary experiments have been modest but the results are promising. We are in the process of negotiating access to additional examples of industrial scale and plan to compare the performance of our method with others reported in the literature. We are also exploring efficient methods for MC/DC coverage.

Our methods use the raw power of modern model checkers. It is likely that analysis of the control flow of the model under examination could target this power more efficiently, and we intend to explore this possibility. Our methods also can use techniques based on abstraction and counterexample-driven refinement, such as those reported by Beyer et al. [2] (ICS, already present as part of SAL, can be used to solve the constraint satisfaction problems), and we intend to examine this combination.

[6] See `vehicle.me.berkeley.edu/mobies/`.

References

[1] R. Alur and D. Peled, editors. *Computer-Aided Verification, CAV '2004*, volume 3114 of *Lecture Notes in Computer Science*, Boston, MA, July 2004. Springer-Verlag.

[2] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *26th International Conference on Software Engineering*, pages 326–335, Edinburgh, Scotland, May 2004. IEEE Computer Society.

[3] P. E. Black and S. Ranville. Winnowing tests: Getting quality coverage from a model checker without quantity. In *20th AIAA/IEEE Digital Avionics Systems Conference*, Daytona Beach, FL, Oct. 2001. Available from `http://hissa.nist.gov/~black/Papers/dasc2001.html`.

[4] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. Technical Report NASA-IVV-96-022, NASA Independent Verification and Validation Facility, Fairmont, WV, Aug. 1996.

[5] L. de Moura, S. Owre, H. Rueß, J. Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In D. Basin and M. Rusinowitch, editors, *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222, Cork, Ireland, July 2004. Springer-Verlag.

[6] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In Alur and Peled [1], pages 496–500.

[7] L. de Moura and H. Rueß. An experimental evaluation of ground decision procedures. In Alur and Peled [1], pages 162–174.

[8] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer-Aided Verification, CAV '2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, Boulder, CO, July 2003. Springer-Verlag.

[9] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering—ESEC/FSE '99: Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162, Toulouse, France, Sept. 1999. Springer-Verlag.

[10] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfstahl. Coverage-directed test generation using symbolic techniques. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 143–158, Palo Alto, CA, Nov. 1996. Springer-Verlag.

[11] G. Hamon and J. Rushby. An operational semantics for Stateflow. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering: 7th International Conference (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243, Barcelona, Spain, 2004. Springer-Verlag.

[12] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A practical tutorial on modified condition/decision coverage. NASA Technical Memorandum TM-2001-210876, NASA Langley Research Center, Hampton, VA, May 2001. Available at `http://www.faa.gov/certification/aircraft/av-info/software/Research/MCDC%20Tutorial.pdf`.

[13] M. P. Heimdahl, Y. Choi, and M. Whalen. Deviation analysis through model checking. In *17th IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 37–46, Edinburgh, Scotland, Sept. 2002. IEEE Computer Society.

[14] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *Third International Workshop on Formal Approaches to Software Testing (FATES)*, volume 2931 of *Lecture Notes in Computer Science*, pages 42–59, Montreal, Canada, Oct. 2003. Springer-Verlag.

[15] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal simulation engines. In *International Conference on Computer Aided Design (ICCAD)*, pages 120–126, Jan Jose, CA, Nov. 2000. Association for Computing Machinery.

[16] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *25th International Conference on Software Engineering*, pages 232–242, Portland, OR, May 2003. IEEE Computer Society.

[17] R. Kelsey, W. Clinger, and J. R. (editors). Revised^5 report on the algorithmic language Scheme. *Higher Order and Symbolic Compututation*, 11(1):7–105, 1998. Available from `http://www.schemers.org/Documents/Standards/R5RS/`.

[18] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes—The Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ, 1994.

[19] The Mathworks. *Stateflow and Stateflow Coder, User's Guide*, release 13sp1 edition, Sept. 2003. Available at `http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf`.

[20] H. Rudin, C. H. West, and P. Zafiropulo. Automated protocol validation: One chain of development. *Computer Networks*, 2:373–380, 1978.

[21] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.